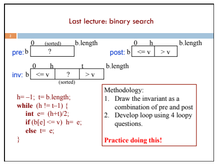
We may not cover all this material

SEARCHING AND SORTING HINT AT ASYMPTOTIC COMPLEXITY

Lecture 9
CS2110 – Spring 2015



Last lecture: binary search

pre: b [0 (sorted) ? b.length]   post: b [0 <= v h > v b.length]

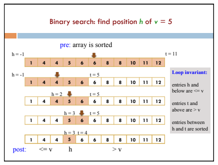
inv: b [0 <= v h ? t > v b.length] (sorted)

```
h= –1;  t= b.length;
while  (h != t–1) {
    int  e=  (h+t)/2;
    if (b[e] <= v)  h=  e;
    else  t=  e;
}
```

Methodology:

1. Draw the invariant as a combination of pre and post
2. Develop loop using 4 loopy questions.

Practice doing this!



Binary search: find position h of v = 5

pre: array is sorted

1	4	4	5	6	6	8	8	10	11	12

h = -1, t = 11

h = -1, t = 5

h = 2, t = 5

h = 3, t = 5

h = 3, t = 4

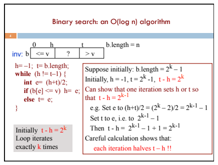
post: <= v h > v

Loop invariant:

entries h and below are <= v

entries t and above are > v

entries between h and t are sorted



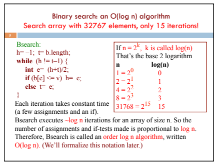
Binary search: an O(log n) algorithm

inv: b [0 <= v h ? t > v] b.length = n

```
h= –1;  t= b.length;
while  (h != t–1) {
    int  e=  (h+t)/2;
    if (b[e] <= v)  h=  e;
    else  t=  e;
}
```

Initially $t - h = 2^k$
Loop iterates exactly k times

Suppose initially: b.length $= 2^k – 1$
Initially, $h = -1$, $t = 2^k -1$, $t - h = 2^k$
Can show that one iteration sets h or t so that $t - h = 2^{k-1}$
e.g. Set e to $(h+t)/2 = (2^k – 2)/2 = 2^{k-1} – 1$
Set t to e, i.e. to $2^{k-1} – 1$
Then $t - h = 2^{k-1} – 1 + 1 = 2^{k-1}$
Careful calculation shows that:
each iteration halves t – h !!



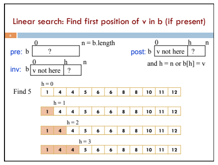
Binary search: an O(log n) algorithm
Search array with 32767 elements, only 15 iterations!

```
Bsearch:
h= –1;  t= b.length;
while  (h != t–1) {
    int  e=  (h+t)/2;
    if (b[e] <= v)  h=  e;
    else  t=  e;
}
```

If $n = 2^k$, k is called log(n)
That's the base 2 logarithm

n	log(n)
$1 = 2^0$	0
$2 = 2^1$	1
$4 = 2^2$	2
$8 = 2^3$	3
$31768 = 2^{15}$	15

Each iteration takes constant time (a few assignments and an if).
Bsearch executes ~log n iterations for an array of size n. So the number of assignments and if-tests made is proportional to log n. Therefore, Bsearch is called an order log n algorithm, written O(log n). (We'll formalize this notation later.)



Linear search: Find first position of v in b (if present)

pre: b [0 ? n = b.length]   post: b [0 v not here h ? n]
and h = n or b[h] = v

inv: b [0 v not here h ? n]

Find 5

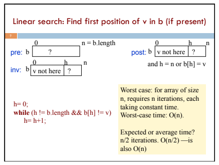
1	4	4	5	6	6	8	8	10	11	12

h = 0

h = 1

h = 2

h = 3



Linear search: Find first position of v in b (if present)

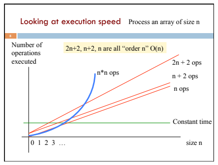
pre: b [0 ? n = b.length]   post: b [0 v not here h ? n]
and h = n or b[h] = v

inv: b [0 v not here h ? n]

```
h= 0;
while (h != b.length && b[h] != v)
    h= h+1;
```

Worst case: for array of size n, requires n iterations, each taking constant time.
Worst-case time: O(n).

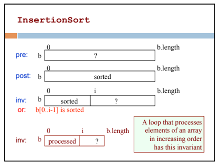
Expected or average time? n/2 iterations. O(n/2) —is also O(n)



Looking at execution speed

Process an array of size n

2n+2, n+2, n are all "order n" O(n)



InsertionSort

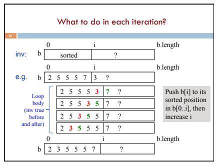
pre: b [0 ? b.length]

post: b [0 sorted b.length]

inv: b [0 sorted i ? b.length]
or: b[0..i-1] is sorted

inv: b [0 processed i ? b.length]

A loop that processes elements of an array in increasing order has this invariant



What to do in each iteration?

inv: b [0 sorted i ? b.length]

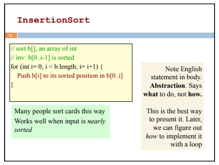
e.g. b [0 2 5 5 5 7 i 3 ? b.length]

Loop body (inv true before and after)

2 5 5 5 3 | 7 ?
2 5 5 3 5 | 7 ?
2 5 3 5 5 | 7 ?
2 3 5 5 5 | 7 ?

Push b[i] to its sorted position in b[0..i], then increase i

b [0 2 3 5 5 5 7 i ? b.length]



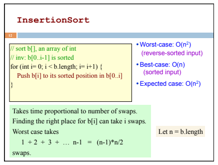
InsertionSort

```
// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i= 0; i < b.length; i= i+1) {
   Push b[i] to its sorted position in b[0..i]
}
```

Many people sort cards this way
Works well when input is *nearly sorted*

Note English statement in body. **Abstraction**. Says **what** to do, not **how.**

This is the best way to present it. Later, we can figure out *how* to implement it with a loop



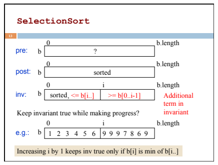
InsertionSort

```
// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i= 0; i < b.length; i= i+1) {
   Push b[i] to its sorted position in b[0..i]
}
```

- Worst-case: $O(n^2)$ (reverse-sorted input)
- Best-case: O(n) (sorted input)
- Expected case: $O(n^2)$

Takes time proportional to number of swaps.
Finding the right place for b[i] can take i swaps.
Worst case takes
1 + 2 + 3 + … n-1 = (n-1)*n/2
swaps.

Let n = b.length



SelectionSort

pre: b [0 ? b.length]

post: b [0 sorted b.length]

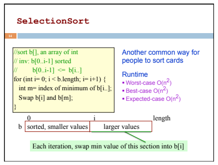
inv: b [0 sorted, <= b[i..] i >= b[0..i-1] b.length]

Additional term in invariant

Keep invariant true while making progress?

e.g.: b [0 1 2 3 4 5 6 i 9 9 9 7 8 6 9 b.length]

Increasing i by 1 keeps inv true only if b[i] is min of b[i..]



SelectionSort

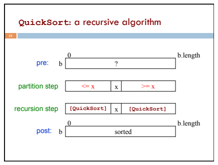
```
//sort b[], an array of int
// inv: b[0..i-1] sorted
//        b[0..i-1]  <=  b[i..]
for (int i= 0; i < b.length; i= i+1) {
   int m= index of minimum of b[i..];
   Swap b[i] and b[m];
}
```

Another common way for people to sort cards

Runtime
- Worst-case $O(n^2)$
- Best-case $O(n^2)$
- Expected-case $O(n^2)$

b [0 sorted, smaller values i larger values length]

Each iteration, swap min value of this section into b[i]



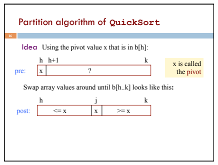
QuickSort: a recursive algorithm

pre: b [0 ? b.length]

partition step [<= x | x | >= x]

recursion step [[QuickSort] | x | [QuickSort]]

post: b [0 sorted b.length]



Partition algorithm of QuickSort

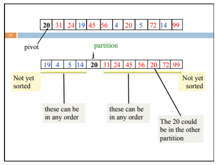
Idea Using the pivot value x that is in b[h]:

pre: [h x h+1 ? k]

x is called the pivot

Swap array values around until b[h..k] looks like this:

post: [h <= x j x >= x k]



20	31	24	19	45	56	4	20	5	72	14	99

pivot

partition

j

19	4	5	14	20	31	24	45	56	20	72	99

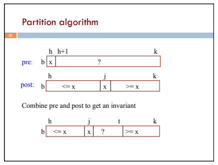
Not yet sorted

these can be in any order

these can be in any order

Not yet sorted

The 20 could be in the other partition



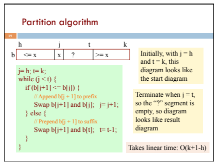
Partition algorithm

pre: b [h x h+1 ? k]

post: b [h <= x j x >= x k]

Combine pre and post to get an invariant

b [h <= x j x ? t >= x k]



Partition algorithm

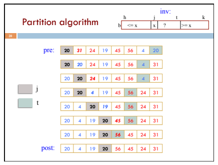
b [h <= x j x ? t >= x k]

```
j= h; t= k;
while (j < t) {
   if (b[j+1] <= b[j]) {
       // Append b[j + 1] to prefix
       Swap b[j+1] and b[j];   j= j+1;
   } else {
       // Prepend b[j + 1] to suffix
       Swap b[j+1] and b[t];   t= t-1;
   }
}
```

Initially, with j = h and t = k, this diagram looks like the start diagram

Terminate when j = t, so the "?" segment is empty, so diagram looks like result diagram

Takes linear time: O(k+1-h)



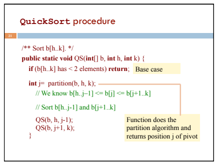
Partition algorithm

inv: b [h <= x j x ? t >= x k]

pre:	20	31	24	19	45	56	4	20
	20	20	24	19	45	56	4	31
	20	20	24	19	45	56	4	31
	20	20	4	19	45	56	24	31
	20	4	20	19	45	56	24	31
	20	4	19	20	45	56	24	31
	20	4	19	20	56	45	24	31
post:	20	4	19	20	56	45	24	31

▨ j

▨ t



QuickSort procedure

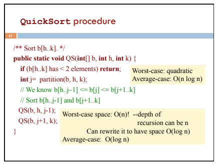
```
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
   if (b[h..k] has < 2 elements) return;     Base case

   int j=  partition(b, h, k);
      // We know b[h..j–1] <= b[j] <= b[j+1..k]

      // Sort b[h..j-1] and b[j+1..k]

      QS(b, h, j-1);
      QS(b, j+1, k);
   }
```

Function does the partition algorithm and returns position j of pivot

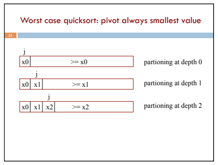


QuickSort procedure

```
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
   if (b[h..k] has < 2 elements) return;
   int j=  partition(b, h, k);
   // We know b[h..j–1] <= b[j] <= b[j+1..k]
   // Sort b[h..j-1] and b[j+1..k]
  QS(b, h, j-1);
  QS(b, j+1, k);
}
```

Worst-case: quadratic
Average-case: O(n log n)

Worst-case space: O(n)! --depth of recursion can be n
Can rewrite it to have space O(log n)
Average-case: O(log n)

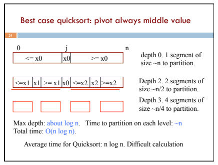


Worst case quicksort: pivot always smallest value

[j x0 | >= x0] partioning at depth 0

[x0 | j x1 | >= x1] partioning at depth 1

[x0 | x1 | j x2 | >= x2] partioning at depth 2



Best case quicksort: pivot always middle value

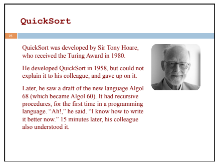
[0 <= x0 | j x0 | >= x0 n] depth 0. 1 segment of size ~n to partition.

[<=x1 | x1 | >= x1 | x0 | <=x2 | x2 | >=x2] Depth 2. 2 segments of size ~n/2 to partition.

Depth 3. 4 segments of size ~n/4 to partition.

Max depth: about log n. Time to partition on each level: ~n
Total time: O(n log n).

Average time for Quicksort: n log n. Difficult calculation

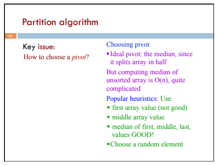


QuickSort

QuickSort was developed by Sir Tony Hoare, who received the Turing Award in 1980.

He developed QuickSort in 1958, but could not explain it to his colleague, and gave up on it.

Later, he saw a draft of the new language Algol 68 (which became Algol 60). It had recursive procedures, for the first time in a programming language. "Ah!," he said. "I know how to write it better now." 15 minutes later, his colleague also understood it.



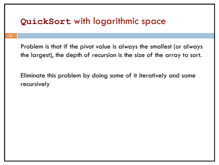
Partition algorithm

Key issue:
How to choose a *pivot*?

Choosing pivot
- Ideal pivot: the median, since it splits array in half

But computing median of unsorted array is O(n), quite complicated

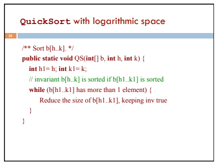
Popular heuristics: Use
- first array value (not good)
- middle array value
- median of first, middle, last, values GOOD!
- Choose a random element



QuickSort with logarithmic space

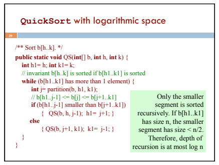
Problem is that if the pivot value is always the smallest (or always the largest), the depth of recursion is the size of the array to sort.

Eliminate this problem by doing some of it iteratively and some recursively



QuickSort with logarithmic space

```
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
   int h1= h; int k1= k;
   // invariant b[h..k] is sorted if b[h1..k1] is sorted
   while (b[h1..k1] has more than 1 element) {
        Reduce the size of b[h1..k1], keeping inv true
   }
}
```



QuickSort with logarithmic space

```
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
   int h1= h; int k1= k;
   // invariant b[h..k] is sorted if b[h1..k1] is sorted
   while (b[h1..k1] has more than 1 element) {
       int j= partition(b, h1, k1);
       // b[h1..j-1] <= b[j] <= b[j+1..k1]
       if (b[h1..j-1] smaller than b[j+1..k1])
           {   QS(b, h, j-1);  h1=  j+1; }
      else
           { QS(b, j+1, k1);  k1=  j-1; }
   }
}
```

Only the smaller segment is sorted recursively. If b[h1..k1] has size n, the smaller segment has size < n/2. Therefore, depth of recursion is at most log n